Name: _____________________________

Instructions:

This is a closed book, closed notes exam. No electronic devices, including calculators, are allowed. You have 110 minutes. There are 7 problems and 14 pages to this exam.

Good Luck!

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (20)</td>
<td></td>
</tr>
<tr>
<td>2 (10)</td>
<td></td>
</tr>
<tr>
<td>3 (10)</td>
<td></td>
</tr>
<tr>
<td>4 (14)</td>
<td></td>
</tr>
<tr>
<td>5 (10)</td>
<td></td>
</tr>
<tr>
<td>6 (6)</td>
<td></td>
</tr>
<tr>
<td>7 (10)</td>
<td></td>
</tr>
<tr>
<td>8 (20)</td>
<td></td>
</tr>
<tr>
<td>Total (100)</td>
<td></td>
</tr>
</tbody>
</table>
1. (20 points) You are given a (tiny) memory system with the following parameters:

- The page size is an unrealistically-small 32 bytes
- The virtual address space for the process in question (assume there is only one) is 1024 pages, or 32 KB total memory
- Physical memory consists of 128 pages, or 4 KB total memory

The system uses a multi-level page table. Each page directory entry (PDE) is 1 byte long. The uppermost bit is a valid bit, and the lower 7 bits point to the physical page of the page table entry.

A page table entry (PTE) follows a similar format. The upper bit is a valid bit, and the lower 7 bits refer to the physical page for that virtual page (if the valid bit is 1).

You are given the value of the Page Directory Base Register (PDBR) which tells you where the page directory is located.

You are also given a complete physical memory dump of all 128 pages.

You are then given a list of virtual addresses to translate.

PDBR: 34 (decimal) [This means the page directory is held in this page]

Translate the following addresses:

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>0x50f5</th>
<th>0x003d</th>
<th>0x0ca6</th>
<th>0x304b</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDE Index</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PDE Contents</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PDE Valid (Y/N)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTE PFN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTE Index</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTE Contents</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTE Valid (Y/N)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PFN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Physical Address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Answer all numeric values in hexadecimal. If there is no valid entry for a given box, put a N/A in the box.
Workspace for question 1:
2. (10 points) Choose whether spinlocks (implemented as described in class) are a good idea or bad idea under the following situations and why do you think it's a good or bad idea.

<table>
<thead>
<tr>
<th>Situation</th>
<th>Good or Bad Idea?</th>
<th>Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct locking for a critical condition on a single processor system</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Correct locking for a critical condition on a multiprocessor system</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fairness when multiple threads are attempting to get the lock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Low contention for the lock 2. Locks are held for a short time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. High contention for the lock 2. Locks are held for a long time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. (10 Points) Consider the following snapshot of a system: Note: Solution differs from this problem. Figure out which one is correct and make them match.

<table>
<thead>
<tr>
<th>Process</th>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P1</td>
<td>2 3 1</td>
<td>2 9 1</td>
<td>2 3 2</td>
</tr>
<tr>
<td>P2</td>
<td>2 0 0</td>
<td>3 4 3</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>0 0 1</td>
<td>3 7 5</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 2 0</td>
<td>2 3 3</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>1 2 1</td>
<td>2 2 2</td>
<td></td>
</tr>
</tbody>
</table>

Part A: Show that the system is in a safe state by demonstrating an order in which the processes may complete.

Part B: If a request from process P4 arrives for (0, 1, 3) can the request be granted immediately? If it can, show the system is in a safe state by demonstrating an order in which the processes may complete.

Part C: If a request from process P3 arrives for (0, 0, 2) can the request be granted immediately? If it can, show the system is in a safe state by demonstrating an order in which the processes may complete. For this problem assume that the system is in the same state as it was before the request in Part B came in.
4. (14 points) Some of the following arrangement of threads and the locks they will grab, can lead to deadlock. Which ones?

<table>
<thead>
<tr>
<th>Thread 1: will try to grab locks 1 and 2 in arbitrary order.</th>
<th>Thread 2: will try to grab locks 1 and 2 in a fixed order, 1 then 2.</th>
<th>Could this deadlock?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1: will try to grab locks 1 and 2 in arbitrary order.</td>
<td>Thread 2: will try to grab locks 1 and 2 in a fixed order, 1 then 2.</td>
<td>Could this deadlock?</td>
</tr>
<tr>
<td>Thread 1: will try to grab locks 1 and 2 in a fixed order, 1 then 2.</td>
<td>Thread 2: will try to grab locks 1 and 2 in a fixed order, 2 then 1.</td>
<td>Could this deadlock?</td>
</tr>
<tr>
<td>Thread 1: will try to grab locks 1 and 2 in a fixed order, 1 then 2.</td>
<td>Thread 2: will try to grab locks 1, 2, and 3 in a fixed order, 1, then 2, then 3.</td>
<td>Could this deadlock?</td>
</tr>
<tr>
<td>Thread 1: will try to grab locks 1 and 2 in arbitrary order.</td>
<td>Thread 2: will try to grab locks 2 and 3 in arbitrary order.</td>
<td>Could this deadlock?</td>
</tr>
<tr>
<td>Thread 1: will try to grab locks 1 and 2 in arbitrary order.</td>
<td>Thread 2: will try to grab locks 2 and 3 in arbitrary order.</td>
<td>Could this deadlock?</td>
</tr>
<tr>
<td>Thread 1: will try to grab locks 1 and 2 in a fixed order, 1 then 2.</td>
<td>Thread 2: will try to grab locks 2 and 3 in a fixed order, 2 then 3.</td>
<td>Could this deadlock?</td>
</tr>
<tr>
<td>Thread 1: will try to grab locks 1 and 2 in a fixed order, 1 then 2.</td>
<td>Thread 2: will try to grab locks 2 and 3 in a fixed order, 2 then 3.</td>
<td>Could this deadlock?</td>
</tr>
<tr>
<td>Thread 1: will try to grab locks 1 and 2 in a fixed order, 1 then 2.</td>
<td>Thread 2: will try to grab locks 1 and 2 in arbitrary order.</td>
<td>Could this deadlock?</td>
</tr>
</tbody>
</table>
5. (10 points) Assume we have a new hardware instruction called the Load-and-Store-Zero, which does
the following atomically (here is C pseudo-code):

```c
int LoadAndStoreZero(int *addr) {
    int old = *addr;
    *addr = 0;
    return old;
}
```

Let's use this instruction to build a spinlock. Complete the following three functions by filling in the
blank lines. You can shorten any calls to the above instruction as LASZ().

```c
void lock_init(int *lock) {
    // What do we initialize our lock to?
    *lock = ______________;
}

void lock_acquire(int *lock) {
    while(______________________________);
}

void lock_release(int *lock) {
    *lock = ______________;
}
```
6. (6 points) The code below provides a "solution" to the producer/consumer problem. This solution uses a single lock ($m$) and two condition variables ($fill$ and $empty$).

Consumer:
while (1) {
    mutex_lock(&m);
    if (numfull == 0) {
        wait(&fill, &m);
    }
    tmp = get();
    signal(&empty);
    mutex_unlock(&m)
}

Producer:
for (i = 0; i < 10; i++) {
    mutex_lock(&m);
    if (numfull == MAX) {
        wait(&empty, &m);
    }
    put();
    signal(&fill);
    mutex_unlock(&m);
}

We've made some changes to the condition variable library:

1. `wait()` is changed so that it never wakes up unless signaled (in other words, there are no spurious wakeups).
2. `signal()` is changed so that it immediately transfers control to a waiting thread (if there is one), thus implicitly passing the lock to the waiting thread as well. Does this change affect the correctness of the producer/consumer solution above?

Explain below why the above code is either correct or incorrect.
7. (10 points) Assume we want to build a semaphore out of locks and condition variables. We have some basic code pieces lying around:

1. while (s <= 0)
2. while (s >= 0)
3. while (s > 0)
4. while (s < 0)
5. while (s == 0)
6. s--;
7. s++;
8. cond_wait(&cond, &lock);
9. cond_signal(&cond);
10. mutex_lock(&lock);
11. mutex_unlock(&lock);

Your job is to implement some different semaphore variants with these code chunks. For example, a rather useless code sequence would be “6, 7”, which would decrement s by 1 (“s–”) and then increment it by 1 (“s++”).

Assume you are implementing a binary semaphore. Thus, you have to implement two routines: sem_wait() and sem_post(). For the parts below, you can just list the code numbers, you don’t have to write out the actual code.

(a) What are the operations you need to implement sem_wait()?

(b) What are the code parts you need to implement sem_post()?

Now assume you need to build a generalized semaphore. The semaphore is initialized with the following code:

```c
sem_init(int value) {
    s = value;
}
```

where “s” is the variable that holds the value of the semaphore.

(c) What are the operations you need to implement sem_wait()?

(d) What are the operations you need to implement sem_post()?
8. (20 points) Given the following code:

```c
int counter = 0;

mythread(void *arg) {
    int i;
    for (i = 0; i < 10000; i++) {
        counter = counter + 1;
    }
    return NULL;
}
```

Assume two threads A and B are running the above thread at the same time. Circle all possible values of counter below after both threads have completed.

A. 5000
B. 10000
C. 15000
D. 20000
E. 25000

The assembly code to update the counter for both threads is shown below. counter is held in memory at address 0x8049a1c.

```
A1: mov 0x8049a1c, %eax
A2: add $0x1, %eax
A3: mov %eax, 0x8049a1c
B1: mov 0x8049a1c, %eax
B2: add $0x1, %eax
B3: mov %eax, 0x8049a1c
```

Show an interleaving of threads A and B that would give a correct update of the counter by both threads. Lists the steps by thread and instruction (e.g., A1, ...).

Show an interleaving of threads A and B that would give an incorrect update of the counter by both threads. Lists the steps by thread and instruction (e.g., A1, ...).

We decided to protect the above code with a semaphore. What does the semaphore need to be initialized to?

Do you need to protect the variable i with the semaphore? Why or why not?